Še o objektnem programiranju

Objektno programiranje je resna tema, ki je ni mogoče kar tako odpredavati v dveh tednih in se ga ni mogoče kar tako naučiti v dveh mesecih. Ker je vaš študij kratek in boste te reči kmalu potrebovali pri drugih predmetih, vam moramo tule predstaviti osnove. Ker vam mora priti stvar v kri, pa ne pričakujmo čudežev. Pri tem predmetu se boste naučili zgolj nekaterih tehnikalij, ne pa vsega, kar v zvezi s tem ponuja Python, vsega, kar ponujajo drugi jeziki in, predvsem, filozofije, ki je za tem.

Dedovanje

Definirajmo razred `Oseba`. Vsaka oseba ima `ime` in `spol` (`Z` ali `M`); oba podatka podamo kot argumenta konstruktorju. Oseba zna pozdravljati: metoda `pozdravi` izpiše `Pozdravljeni, jaz sem {}`, kjer namesto `{}` vstavi svoje ime.

```
class Oseba:
    def __init__(self, ime, spol):
        self.ime = ime
        self.spol = spol

    def pozdravi(self):
        print("Pozdravljeni, jaz sem {}".format(self.ime))
```

Tu smo mimogrede prvič srečali konstruktor (metodo `__init__`) z argumenti. Želva je imela konstruktor brez argumentov; novo želvo smo naredili s `Turtle()` in vedno je stala na sredini, obrnjena navzgor, vidna in s spuščenim peresom. Pri osebah ne bomo imeli "začetnega imena" (Janez Novak?!), temveč bomo morali ob konstrukciji objekta `Oseba` vedno povedati tudi njeno ime in spol.

```
benjamin = Oseba("Benjamin", "M")
```

Vse, kar naredi konstruktor, je, da prepiše vrednost argumenta v (istoimenski) atribut objekta. Se pravi, shrani argument v objekt.

Osebe znajo tudi vljudno pozdravljati:

```
benjamin.pozdravi()
```

```
Pozdravljeni, jaz sem Benjamin
```

Na fakulteti sta, poenostavljeno povedano, dve vrsti oseb, študenti in učitelji. Za študente beležimo njihove ocene pri posameznih predmetih. Ocene bomo pisali v slovar, katerega ključi bodo imena predmetov, vrednosti pa ocene pri teh predmetih. Študenta je mogoče oceniti, zato bo imel metodo `oceni`, ki ji bomo kot argument podali ime predmeta in oceno, pa bo zapisala to v slovar. Poleg tega bo imel funkcijo `poprecje`, ki bo izračunala njegovo poprečno oceno.

Poleg tega pa imajo tudi študenti ime in spol, in tudi pozdravljati znajo. Z drugimi besedami: *študent je vrsta osebe*. Študent ima vse lastnosti osebe in

zna vse, kar znajo osebe, poleg tega pa še nekaj več (beležiti ocene in računati poprečja). Zato bomo razred `Student` *izpeljali iz razreda* `Oseba` (Angleži bi rekli `Student` *is derived from* `Oseba`). `Student` bo *podedoval* vse, kar ima `Oseba` (`Student` *inherits attributes and methods from* `Oseba`). Razred `Oseba` bo prednik (*parent*) razreda `Student`.

Kako to storimo? Zelo preprosto: ko definiramo razred `Student`, v oklepaju dodamo še razred, iz katerega naj bo ta izpeljan.

```
class Student(Oseba):
    def __init__(self, ime, spol):
        super().__init__(ime, spol)
        self.ocene = {}

    def oceni(self, predmet, ocena):
        self.ocene[predmet] = ocena

    def poprecje(self):
        if not self.ocene:
            return 0
        return sum(self.ocene.values()) / len(self.ocene)
```

Najprej preverimo, ali reč deluje.

```
rebeka = Student("Rebeka", "Z")
rebeka.oceni("Programiranje 1", 10)
rebeka.oceni("Uvod v racunalnistvo", 9)
rebeka.poprecje()
```

```
9.5
```

Sestavimo tri študente, zložimo jih v seznam in vsakemu študentu dodelimo deset naključnih ocen pri naključno izbranih predmetih.

```
from random import choice, randint

s1 = Student("Ana Karenina", "Z")
s2 = Student("Berta Novak", "Z")
s3 = Student("Cilka Celarec", "Z")

studenti = [s1, s2, s3]

predmeti = ["Uvod v racunalnistvo", "Programiranje 1", "Racunalniska arhitektura",
            "Matematika", "Diskretne strukture", "Programiranje 2",
            "Podatkovne baze", "Racunalniske komunikacije", "Operacijski sistemi",
            "Osnove verjetnosti in statistike"]

for student in studenti:
    for i in range(10):
```

```
        student.oceni(choice(predmeti), randint(6, 10))
```

Zdaj lahko izpišemo poprečne ocene vseh študentov.

```
for student in studenti:
    print(student.ime, student.poprecje())
```

```
Ana Karenina 8.142857142857142
Berta Novak 8.0
Cilka Celarec 7.571428571428571
```

Vse to so bile študentske zadeve. Ker je Rebeka tudi oseba, pa zna tudi pozdravljati.

```
rebeka.pozdravi()
```

```
Pozdravljeni, jaz sem Rebeka
```

Še enkrat preverite in se prepričajte: metode `pozdravi` nismo definirali v razredu `Student`, Rebeka pa vseeno pozdravlja. Ta metoda je torej podedovana od razreda-prednika `Oseba`.

Zdaj pa poglejmo definicijo metod razreda. Dodeljevanje ocene (`oceni`) je trivialno. Pri računanju poprečja (`poprecje`) moramo paziti le na študente, ki nimajo še nobene ocene, zato bo `vseh` tam enak 0.

Zanimiv pa je konstruktor, ki mora le pripraviti slovar, v katerega bo metoda `oceni` dodajala ocene. Ostalo pa prepusti podedovanemu konstruktorju - tako da ga pokliče.

Razred `Student` je podedoval, recimo, metodo `pozdravi`. Ko rečemo `rebeka.pozdravi()`, se pokliče podedovano pozdravljanje. `Student` je sestavil novo metodo `poprecje` in tudi tu ni dilem: ko rečemo `rebeka.poprecje()`, se pokliče funkcija oceni iz razreda `Student`. Metodi `__init__` pa sta dve, podedovana in nova! Pravilo je preprosto: istoimenska metoda iz izpeljanega razreda povozi podedovano metodo.

Da je res tako, se najprej prepričajmo na preprostem zgledu: razredu `Student` definirajmo novo metodo `pozdravi`.

```
class Student(Oseba):
    def __init__(self, ime, spol):
        super().__init__(ime, spol)
        self.ocene = {}

    def oceni(self, predmet, ocena):
        self.ocene[predmet] = ocena

    def poprecje(self):
        if not self.ocene:
            return 0
        return sum(self.ocene.values()) / len(self.ocene)
```

```
    def pozdravi(self):
        print("Pozdravljeni, jaz sem {} in sem student(ka)".format(self.ime))
```

Sestavimo novo Rebeko in jo pozovimo, naj nas pozdravi.

```
rebeka = Student("Rebeka", "Z")
rebeka.pozdravi()
```

```
Pozdravljeni, jaz sem Rebeka in sem student(ka)
```

Rebeka ima tule dve metodi `pozdravi`, vendar se pokliče nova in ne podedovana.

Kaj pa, če bi hoteli poklicati podedovano metodo in ne nove? Tega ne počnemo. To je nenaravno, ne predstavljam si situacije, kjer bi nam v lepo napisanem programu to moglo priti prav ... razen v enem primeru: ko nova metoda kliče staro. Tule lahko razmišljamo takole: `Student` je izpeljan iz `Oseba` in torej zna pozdravljati. Vse, kar doda k pozdravu, je "in sem student(ka)". To pomeni, da bi lahko najprej prepustili pozdravljanje podedovani metodi in v `Student`ovem `pozdravi` le še dodatno izpisali "in sem student(ka)". To se stori takole:

```
class Student(Oseba):
    def __init__(self, ime, spol):
        super().__init__(ime, spol)
        self.ocene = {}

    def oceni(self, predmet, ocena):
        self.ocene[predmet] = ocena

    def poprecje(self):
        if not self.ocene:
            return 0
        return sum(self.ocene.values()) / len(self.ocene)

    def pozdravi(self):
        super().pozdravi()
        print("in sem student(ka)")
```

Rezultat sicer ni povsem enak ("in sem student(ka)" je namreč napisano v novi vrsti), a za primer bomo to vzeli v zakup.

`super()` je čudna funkcija, ki nam - precej poenostavljeno - vrne `self` *kot da bi bil* objekt razreda, ki je prednik razreda `Student` (torej razreda `Oseba`). Torej, če rečemo `self.pozdravi()` se pokliče metoda `pozdravi`, ki smo jo definirali v razredu `Student`, saj je `self` objekt razreda `Student`. Če pa pokličemo `super().pozdravi()`, se pokliče metoda `pozdravi` razreda `Oseba`, ker je razred `Oseba` prednik razreda `Student`. (Za tiste, ki veste, kaj je večkratno dedovanje: funkcija `super` v resnici vrača nekaj veliko veliko bolj zapletenega, saj je potrebno, kadar dedujemo iz večih razredov, vsakič poiskati, kateri od podedovanih razredov

vsebuje funkcijo, ki jo želimo klicati. Če hočete moder nasvet: večkratno dedovanje je gladko tlakovana široka pot v pekel. Izogibajte se ga.)

Zdaj lahko končno pogledamo, kaj naredi konstruktor. Poleg `self` dobi še dva argumenta, `ime`, `spol`. Zanju poskrbi stari, podedovani konstruktor, zato ga pokličemo, takole: `super().__init__(ime, spol)`. V novem konstruktorju je potrebno le še pripraviti slovar z ocenami.

Pa definirajmo še razred `Ucitelj`. Tudi ta je `Oseba` (ima ime in starost in zna pozdravljati). Ena od njegovih lastnosti je seznam predmetov, ki jih predava.

```
class Ucitelj(Oseba):
    def __init__(self, ime, spol, predmeti):
        super().__init__(ime, spol)
        self.predmeti = predmeti

u1 = Ucitelj("Tomaž Poljanšek", "M", ["Programiranje 2"])
```

Konstruktor nas, upam, ne preseneča več: `ime` in `spol` prepusti podedovanemu konstruktorju, sam pa le še nastavi seznam predmetov.

Kar je, je; česar ni, pa ni

Sestavimo funkcijo, ki izpiše imena vseh oseb na podanem seznamu.

```
def izpisi_imena(s):
    for e in s:
        print(e.ime)
```

Pokličemo jo lahko s seznamom študentov.

```
izpisi_imena(studenti)
```

```
Ana Karenina
Berta Novak
Cilka Celarec
```

Pokličemo jo lahko tudi s seznamom, v katerem so tako učitelji kot študenti.

```
ljudje = [s1, s2, s3, u1]
izpisi_imena(ljudje)
```

```
Ana Karenina
Berta Novak
Cilka Celarec
Tomaž Poljanšek
```

Pa če imamo kar nek seznam?

```
s = [1, 2, 3]
izpisi_imena(s)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-16-4fcdfb3963eb> in <module>
      1 s = [1, 2, 3]
----> 2 izpisi_imena(s)

<ipython-input-13-207846f9e46f> in izpisi_imena(s)
      1 def izpisi_imena(s):
      2     for e in s:
----> 3         print(e.ime)

AttributeError: 'int' object has no attribute 'ime'
```

```
s = ["Ana", "Berta", "Cilka"]
izpisi_imena(s)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-17-54c61afa8ccb> in <module>
      1 s = ["Ana", "Berta", "Cilka"]
----> 2 izpisi_imena(s)

<ipython-input-13-207846f9e46f> in izpisi_imena(s)
      1 def izpisi_imena(s):
      2     for e in s:
----> 3         print(e.ime)

AttributeError: 'str' object has no attribute 'ime'
```

Ne in ne. Ne s seznamom števil ne s seznamom nizov ni zadovoljna. Čemu ne? No, saj pove: `'int' object has no attribute 'ime'`. Nekje v funkciji piše `e.ime`, pri čemer je `e` element seznama. V prvem primeru to pomeni, da hočemo poklicati `1.ime`, kar je očitno traparija. V drugem hočemo klicati, recimo, `"Ana".ime`, kar prav tako ne gre. Ana sicer ni brez vsega, ima celo kup metod - `replace`, `upper`, `endswith` in še desetine drugih - a metode oz. atributa `ime` ni med njimi.

Funkcija `izpisi_imena` ima torej določene zahteve: kot argument ji moramo dati seznam (točneje: nekaj, prek česar je mogoče spustiti zanko `for`, recimo seznam) in elementi tega seznama morajo biti objekti, ki imajo polje `ime`. Z drugimi besedami: biti morajo objekti tipa `Oseba`.

Res? Pravzaprav ne. Zadošča, da imajo polje `ime`, ni pa treba, da so osebe.

```
class KrNeki:
    def __init__(self):
        self.ime = "jazbec"

a, b, c = KrNeki(), KrNeki(), KrNeki()
```

```
a
```

```
<__main__.KrNeki at 0x7f8bc08266d0>
```

```
a.ime
```

```
'jazbec'
```

```
izpisi_imena([a, b, c])
```

```
jazbec
jazbec
jazbec
```

(Tisti, ki so vajeni takoimenovanih statično tipiziranih jezikov, se zgražajo: funkcija `izpisi_imena` bi morala povedati, kakšnega tipa morajo biti objekti, ki jih sprejme. Tisti, ki imamo radi *tudi* dinamično tipizirane jezike, pa razumemo, zakaj je včasih boljše eno, včasih drugo.)

Podobno kot Python se obnašajo tudi drugi jeziki iste sorte, na primer JavaScript (kjer so reči še bolj drastične, saj imamo objektov, nimamo pa razredov!). Temu slogu (ne)preverjanja tipov pravimo duck typing: *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.* Pri tem ni pomembno, ali gre v resnici za raco ali ne; če se objekt vede kot raca, ga lahko obravnavamo kot raco.

Navidezne metode

S slednjim je povezana še ena reč, ki je v dinamičnih jezikih precej drugačna kot v statično tipiziranih. (Tisti, ki nimate pojma, o čem govorim, se boste naučili, kako je to v Pythonu in se v drugem semestu čudili Javi. Tisti, ki poznate Javo ali C++ ali C#, pa se boste zdajle čudili Pythonu.)

Recimo, da bi imel metodo `naziv`, ki bi vrnila naziv osebe. Naziv osebe je lahko `profesor`, lahko pa je tudi `student` ali `studentka`. Če bi imeli takšno funkcijo, bi lahko pozdrav napisali takole.

```
    def pozdravi(self):
        print("Pozdravljeni, jaz sem {} {}".format(self.naziv(), self.ime))
```

Namreč, pred `ime` smo vrinili še `self.naziv()`, metodo, ki je (še?) nimamo in ki bo vrnila naziv.

No, pa naredimo, kot smo sklenili: spremenimo metodo `pozdravi` (obenem pa poenostavimo razred `Student`, odvzemimo mu ocene, zato pa tudi ne potrebuje več konstruktorja in ker v tem trenutku nima nobene metode, nekaj pa je znotraj razreda vseeno potrebno napisati, napišimo `pass`). Poglejmo, kaj se zgodi.

```
class Oseba:
    def __init__(self, ime, spol):
        self.ime = ime
```

```
        self.spol = spol

    def pozdravi(self):
        print("Pozdravljeni, jaz sem {} {}".format(self.naziv(), self.ime))

class Student(Oseba):
    pass
```

```
rebeka = Student("Rebeka", "Z")
```

```
rebeka.ime
```

```
'Rebeka'
```

```
rebeka.pozdravi()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-25-04d7d0f7c1c2> in <module>
----> 1 rebeka.pozdravi()

<ipython-input-22-80b095f344f1> in pozdravi(self)
      5
      6     def pozdravi(self):
----> 7         print("Pozdravljeni, jaz sem {} {}".format(self.naziv(), self.ime))
      8
      9 class Student(Oseba):

AttributeError: 'Student' object has no attribute 'naziv'
```

Rebeko smo uspeli narediti, tudi ime ima, vendar pa javi napako, ko ji rečemo, naj pozdravi: za to bi potrebovala metodo `naziv()`, te pa ni.

Metodo `naziv` pa bomo naredili ločeno za študente in za učitelje.

```
class Oseba:
    def __init__(self, ime, spol):
        self.ime = ime
        self.spol = spol

    def pozdravi(self):
        print("Pozdravljeni, jaz sem {} {}".format(self.naziv(), self.ime))

class Student(Oseba):
    def naziv(self):
        if self.spol == "Ž":
            return "študentka"
        else:
            return "študent"
```

```
class Ucitelj(Oseba):
    def naziv(self):
            if self.spol == "Z":
                return "profesorica"
            else:
                return "profesor"
```

Pa poglejmo, kako deluje.

```
s1 = Student("Ana Karenina", "Ž")

s1.pozdravi()
```

```
Pozdravljeni, jaz sem študentka Ana Karenina
```

```
u1 = Ucitelj("Tomaz Poljanšek", "M")

u1.pozdravi()
```

```
Pozdravljeni, jaz sem profesor Tomaz Poljanšek
```

Če pomislimo, kaj se tule v resnici dogaja, je stvar nekoliko fascinantna. Poklicali smo metodo `pozdravi`. Metoda `pozdravi` je podedovana iz `Oseba` - ne `Student` ne `Ucitelj` nimata svoje metode `pozdravi`. Izvaja se torej `pozdravi`, ki je definirana v `Oseba`. In med izvajanjem te metode pokličemo metodo `self.naziv`. V prvem primeru (Ana) je `self` objekt razreda `Student`. Razred `Student` ima metodo `naziv` in ko `pozdravi` pokliče `self.naziv` se torej pokliče `Student`ov naziv. V drugem primeru (Tomaz) je `self` objekt razreda `Ucitelj`, torej je `self.naziv` `Uciteljev` `naziv`. Torej: ko `pozdravi` pokliče `self.naziv` to včasih pomeni funkcijo naziv, ki je definirana v `Student`, včasih pa funkcijo `naziv`, ki je definirana v `Ucitelj`.

Tako kot je funkcija `izpisi_imena` zahtevala, da imajo vsi objekti na seznamu `ime`, tako metoda `pozdravi` zahteva, da ima `self` metodo `naziv` (ki ne sprejme drugega argumenta kot `self` in ki vrača nekaj, kar se da izpisati...). Odkod se ta `naziv` vzame - je podedovan, je na novo definiran v `self`ovem razredu ali pa je prišel v `self` po kakih drugih, sumljivejših poteh - ji je vseeno. Pomembno je le, da `self.naziv` obstaja.

Abstraktni razredi

Kar smo povedali ravnokar, je res res pomembno. To je najpomembnejša reč v objektnem programiranju, zato jo moramo še malo raziskovati.

```
nekdo = Oseba("Akakij Akakijevič", "M")

nekdo.pozdravi()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
```

```
<ipython-input-29-a47a23fbd476> in <module>
      1 nekdo = Oseba("Akakij Akakijevič", "M")
      2
----> 3 nekdo.pozdravi()

<ipython-input-26-f11e22da7a6f> in pozdravi(self)
      5
      6     def pozdravi(self):
----> 7         print("Pozdravljeni, jaz sem {} {}".format(self.naziv(), self.ime))
      8
      9 class Student(Oseba):

AttributeError: 'Oseba' object has no attribute 'naziv'
```

To smo v bistvu že videli zgoraj: študenti niso znali pozdravljati, dokler jim nismo priskrbeli metode `naziv`. A to v bistvu pomeni, da je napaka v razredu `Oseba`, ker zahteva nekaj, česar (potencialno) ni.

Temu se ne reče napaka. Temu se reče abstrakten razred. V nekaterih jezikih bi bilo potrebno "napovedati" metodo `naziv` kot "čisto navidezno metodo" (*pure virtual method*). Razredu `Oseba` bi rekli, da je abstrakten (*abstract class*) in objektov tega razreda ne bi bilo mogoče sestavljati - napako bi dobili že ob klicu `Oseba("Akakij Akakijevič", "M")`, češ da takšnih, nepopolnih objektov ne sme biti.

V Pythonu in podobnih jezikih ni tako. Razred `Oseba` je okrnjen, vendar bi lahko načelno imel tudi kakšne metode, ki bi normalo delovale. Študenti bi lahko še vedno prejemali ocene, le pozdravljali ne bi.

Navidezne metode - še enkrat

Nadaljujmo: dodajmo še en razred. Ali dva.

```
class Snazilka(Oseba):
    def __init__(self, ime):
        super().__init__(ime, "Ž")

class Hisnik(Oseba):
    def __init__(self, ime):
        super().__init__(ime, "M")

fata = Snazilka("Fata")

fata.pozdravi()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-30-6c9c721ce1f8> in <module>
```

```
      9 fata = Snazilka("Fata")
     10
---> 11 fata.pozdravi()

<ipython-input-26-f11e22da7a6f> in pozdravi(self)
      5
      6     def pozdravi(self):
----> 7         print("Pozdravljeni, jaz sem {} {}".format(self.naziv(), self.ime))
      8
      9 class Student(Oseba):

AttributeError: 'Snazilka' object has no attribute 'naziv'
```

Da `Snazilka` in `Hisnik` ne znata pozdravljati, je jasno in pričakovano. Ne, ker ne bi znala, le ne vesta, kako bi se predstavila. Podobno bi se zgodilo, ko bi dodali tajnico in osebje študentskega referata.

V vsakem jeziku in družbi pa se razvijejo norme naslavljanja neznancev. V davnih časih moje mladosti se je neznancem uradno reklo tovariš in tovarišica (v praksi se je to nanašalo samo na učitelje in učiteljice), danes sta to gospod in gospa. To so privzeti nazivi, odvisno od situacije pa uporabimo drugačnega (doktor, če gre za zdravnika ali koga, ki se je potrudil napisati daljše besedilo, ki je zaradi vezave v črno usnje dajalo vtis pomembnosti; docent, izredni profesor ali profesor za nekoga, ki je naredil vse potrebne kljukice in tako naprej). Vsi, ostali, ki niso nič posebnega, pa so torej gospodje in gospe, zato opremimo `Osebo` s tem, privzetim nazivom.

```
class Oseba:
    def __init__(self, ime, spol):
        self.ime = ime
        self.spol = spol

    def pozdravi(self):
        print("Pozdravljeni, jaz sem {} {}.".format(self.naziv(), self.ime))

    def naziv(self):
        if self.spol == "Ž":
            return "gospa"
        else:
            return "gospod"

class Student(Oseba):
    def naziv(self):
        if self.spol == "Ž":
            return "študentka"
        else:
            return "študent"
```

```
class Ucitelj(Oseba):
    def naziv(self):
            if self.spol == "Z":
                return "profesorica"
            else:
                return "profesor"

class Snazilka(Oseba):
    def __init__(self, ime):
        super().__init__(ime, "Ž")

class Hisnik(Oseba):
    def __init__(self, ime):
        super().__init__(ime, "M")
```

Poselimo fakulteto.

```
ana = Student("Ana Karenina", "Ž")
tomaz = Ucitelj("Tomaž Poljanšek", "M")
fata = Snazilka("Fata")
joze = Hisnik("Jože")
```

Vsi so vljudni in se lepo pozdravljajo.

```
ana.pozdravi()
```

```
Pozdravljeni, jaz sem študentka Ana Karenina.
```

```
tomaz.pozdravi()
```

```
Pozdravljeni, jaz sem profesor Tomaž Poljanšek.
```

```
fata.pozdravi()
```

```
Pozdravljeni, jaz sem gospa Fata.
```

```
joze.pozdravi()
```

```
Pozdravljeni, jaz sem gospod Jože.
```

`ana.pozdravi()` pokliče Osebin `pozdravi` (temu bomo poslej rekli `Oseba.pozdravi`; to v resnici obstaja, Pythonu lahko zares napišemo `Oseba.pozdravi`, a to, kar v tem primeru dobimo, je nekoliko težko razložiti). Torej `ana.pozdravi()` pokliče metodo `Oseba.pozdravi`. Le-ta kliče `self.naziv`. Ker je `self` isto kot `ana`, `ana` pa je *objekt tipa* (ali, kot sem govoril prej, *razreda* - stvar navade) `Student`, `self.naziv` pomeni `ana.naziv`, torej `Student.naziv`. Rezultat je niz "studentka".

`dani.pozdravi()` pokliče `Oseba.pozdravi`. Le-ta spet kliče `self.naziv`. Ker je `self` isto kot `dani`, `dani` pa je *objekt tipa* `Ucitelj`, `self.naziv` pomeni `Ucitelj.naziv`. Rezultat je niz "profesor", saj `dani.naziv()` vrne `"profesor"`.

And now for something completely different. `fata.pozdravi()` pokliče `Oseba.pozdravi`. Le-ta spet kliče `self.naziv`. Ker je `self` isto kot `fata`, `fata` pa je *objekt tipa* `Snazilka`, `self.naziv` pomeni `Snazilka.naziv`. Tega pa ni! Pač pa ima `fata` naziv, ki ga je podedovala od `Oseba`, namreč `Oseba.naziv`. In ta pravi, da je Fata gospa.

O Nepythonu

Če želimo na tem predavanju vsaj omeniti tudi objektno programiranje na splošno, ne le v Pythonu, moramo povedati za bistveno razliko med objektnim programiranjem v Pythonu (in drugih tipičnih dinamično tipiziranih jezikih (se opravičujem za besednjak), na primer JavaScriptu) in objektnim programiranjem v statično tipiziranih jezikih, na primer C++, Javi in C#.

Metodam, kakršen je tale `naziv`, pravimo *navidezne metode* (*virtual method*). Za navidezne metode je značilno, da imajo različni podedovani razredi lahko različne metode, pri čemer pa predniki "vedo", katero metodo poklicati. Tako `Oseba.pozdrav` včasih pokliče `Oseba.naziv`, včasih `Student.naziv` in včasih `Ucitelj.naziv`. Če bi se pojavil še kak četrti, peti, ali osmi razred s svojim nazivom, bi znal `Oseba.pozdrav` poklicati tudi tega.

V Pythonu izraz sam nima pomena (čemu "navidezna"?!), pa tudi govoriti o navideznih metodah v Pythonu nima veliko smisla, saj drugačnih metod kot navideznih v Pythonu sploh ni. V večini statično tipiziranih jezikih pa obstajajo tudi "nenavidezne" metode. Če `naziv` ne bi bil navidezna metoda, bi `Oseba.pozdrav` vedno poklical `Oseba.naziv`, četudi bi imel objekt, s katerim imamo opravka, svoj, "specializiran" `naziv`.

Drugo, po čemer se ti jeziki razlikujejo od Pythona (in Javascripta in podobnih), je, da bi, kot smo že omenili, zahtevali, da je `Oseba.naziv` nujno definiran (točneje: deklariran), če hočemo, da `Oseba.pozdrav` kliče `self.naziv` - ne glede na to, ali bi bila to navidezna metoda ali ne. Včasih se zgodi tudi, da funkcije, kakršna je `Oseba.naziv`, sploh ne moremo napisati (ker bi morala delati kaj, česar se z `Osebo` ne da narediti, temveč lahko to delajo šele njeni nasledniki). V tem primeru, uh, bi definirali čisto navidezno metodo (*pure virtual method*), razred `Oseba` bi bila abstraktni razred ... No, boljše, da ne rinemo v to. Vedite le, da nam je v Pythonu prihranjenih veliko komplikacij.

Na predavanjih pri Programiranju 1 objektno programiranje v Pythonu popraskamo le po površju. Veliko konceptov, kot na primer statične metode, spremenljivke razreda in podobno, ki ste jih morda srečali v drugih jezikih, pozna tudi Python. Vendar je vse to še pretežko za večino poslušalcev tega predmeta. Te stvari boste spoznavali sproti. Še več pa je tehnik, ki so specifične za Python ali pa jih poznajo le skriptni jeziki. Tudi za te pri Programiranju 1 žal ne bo časa.